Octans: Optimal Placement of Service Function Chains in Many-Core Systems

Zhilong Zheng[†], Jun Bi[†], Heng Yu[†], Haiping Wang[†], Chen Sun[†], Hongxin Hu[§], Jianping Wu[†]
[†]Institute for Network Sciences and Cyberspace, Tsinghua University
[†]Department of Computer Science, Tsinghua University
[†]Beijing National Research Center for Information Science and Technology (BNRist)
[§]School of Computing, Clemson University

***Abstract*—Network Function Virtualization (NFV) has the potential to offer service delivery flexibility and reduce overall costs by running service function chains (SFCs) on commodity servers with many cores. Existing solutions for placing SFCs in one server treat all CPU cores as equal and allocate isolated CPU cores to different network functions (NFs). However, advanced servers often adopt Non-Uniform Memory Access (NUMA) architecture to improve the scalability of many-core systems. CPU cores are grouped into nodes, incurring performance bottleneck due to cross-node memory access and intra-node resource contention. Our evaluation shows that randomly selecting cores to place NFs in an SFC could suffer from 39.2% lower throughput comparing to an optimal placement solution. In this paper, we propose Octans, an NFV orchestrator to achieve maximum aggregate throughput of all SFCs in many-core systems. Octans first formulates the optimization problem as a Non-Linear Integer Programming (NLIP) model. Then we identify the key factor for problem solving as evaluating the throughput drop of an NF caused by other NFs in the same SFC or different SFCs, i.e. *performance drop index*, and propose a formal and precise prediction model based on system level performance metrics. Finally, we propose an efficient heuristic algorithm to quickly find near-optimal placement solutions. We have implemented a prototype of Octans. Extensive evaluation shows that Octans significantly improves the aggregate throughput comparing to two state-of-the-art placement mechanisms by 26.7%∼51.8%, with very low prediction errors of SFC performance (an average deviation of 2.6%). Moreover, Octans could quickly find a near-optimal placement solution with tiny optimality gap (1.2%∼3.5%).**

I. Introduction

Network Function Virtualization (NFV) was recently introduced to address the limitations of traditional middleboxes. NFV runs network functions (NFs) on commodity servers with general-purpose processors such as Intel x86 to improve service delivery flexibility and reduce overall costs. In NFV, packets are usually processed by a sequence of NFs, which form a *service function chain (SFC)*.

Nowadays, commodity servers used in NFV are high-performance and high-density with *multiple CPU cores* [1], which we refer to as *many-core systems*. One such server has the capability of accommodating an entire SFC or even multiple SFCs [2]. In this situation, current solutions for the *placement* of NFs in one server is to treat all CPU cores as equal, and allocate isolated CPU cores to different NFs, in order to avoid performance affection between NFs [2], [3].

However, above solutions overlook the fact that CPU cores in a many-core system are actually *unequal*. Using the different core sets to support the same SFC could result in significantly different throughput. This is because many-core systems today usually adopt Non-Uniform Memory Access (NUMA) architecture for high scalability [4]. Fig. 1 shows a typical architecture of the Intel x86 many-core system. CPU cores are grouped into *nodes*. Each node contains multiple cores and its own local memory. Randomly selecting CPU cores to support an SFC could suffer from seriously compromised throughput due to the following two reasons.

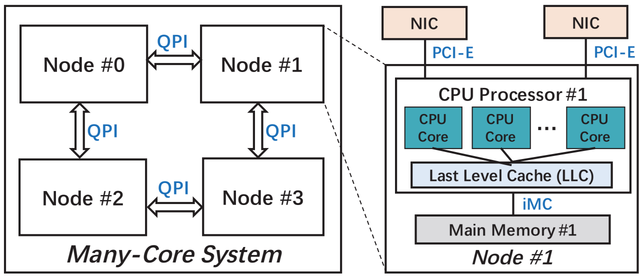
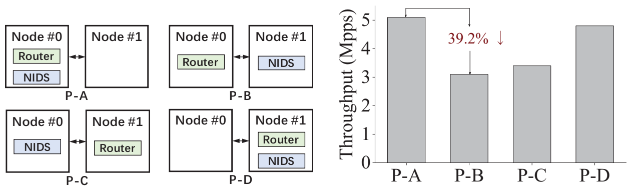


Fig. 1: A high-level view of a typical many-core system.

(1) *Bottleneck incurred by cross-node memory access.* While CPU cores in one node can access the memory in another node via Intel QuickPath Interconnect (QPI), local memory access inside one node is much faster than remote access. To study its effect on the performance of NFV, we place a simple SFC (*Router* $\rightarrow$ *NIDS*) on a many-core system with two nodes in four ways shown in Fig. 2(a) and evaluate their performance. As illustrated in Fig. 2(b), the performance of worst-case placement (i.e., P-B) achieves less throughput than the best-case placement (i.e., P-A) by 39.2%. An intuitive solution is to place all NFs in an SFC in the same node to avoid remote memory access. However, the number of cores in one node is limited. An SFC may be placed across multiple nodes for better resource utilization [1]. Moreover, we can observe from Fig. 2(b) that even switching the node assignment of two NFs (P-B and P-C) could lead to different SFC performances. This is because different NFs expose different performance sensitivity to cross-node memory access.

(2) *Bottleneck incurred by intra-node resource contention.* As shown in Fig. 1, CPU cores in each node may contend to shared resources such as last-level cache (LLC), integrated memory controller (iMC), and QPI. Some recent research efforts [5]–[7] have revealed that co-locating multiple NFs in the same node could decrease the throughput of a single NF



(a) Four ways of SFC placement. (b) SFC performance

Fig. 2: Effect of remote memory access on SFC throughput. The Network Interface Card (NIC) is connected to Node #0. So we call Node #0 *local node*, and Node #1 *remote node*.

by 12.36% to 50.3% due to resource contention.

Above observations motivate us to design an optimal mechanism to place multiple SFCs in a many-core system, with the goal of achieving *maximized aggregate throughput of all SFCs*. Many research efforts have been devoted to addressing the placement problem in NFV [8]–[10]. However, they all focused on finding right servers for SFC placement, while ignoring the placement inside one server. Meanwhile, the problem of placing multiple threads in many-core systems to achieve optimal execution performance has been well studied in the system and architecture community [11]–[14]. However, they relied on frequent migration of threads for optimality, while migrating NFs in the NFV context brings significant performance overhead [15]. We present more details in § II.

In this paper, we propose Octans, an NFV orchestrator for optimal SFC placement inside one server. To understand the key factors in the optimization problem, we start by formulating the problem with a Non-Linear Integer Programming (NLIP) model. Then we identify the key factor for problem solving as evaluating how much the throughput of an NF is affected by NFs in the same SFC or other SFCs, which we refer to as *performance drop index*. This task is challenging in two aspects: (1) to evaluate the throughput drop of NFs caused by resource contention, we are challenged to find a *unified* set of system-level metrics that represent the performance of *massive and heterogeneous NFs* in NFV; and (2) the NFs with different chaining methods could introduce different types of shared resources, which will affect the aggregate performance of SFCs. We are challenged to precisely model NF performance under the SFC context. To address above challenges, we use a formal approach to find performance metrics and construct a comprehensive model for accurate performance drop prediction. Finally, due to the NP-hardness of our problem, we propose a heuristic algorithm to quickly find an optimal or a near-optimal solution.

Octans makes the following major contributions:

- We identify the problem of optimal SFC placement in a many-core system, and present Octans, an NFV orchestrator, to maximize aggregate throughput of SFCs. We introduce related work and highlight the novelty of Octans (§ II).
- We formulate the optimization problem using an NLIP model (§ III). To evaluate the performance drop index due to resource contention and chaining, we introduce a formal approach to find performance metrics, and present an accurate model for performance drop prediction. Finally, we design an online placement algorithm to efficiently produce an optimal or near-optimal solution. (§ IV)
- We introduce the architecture and workflow of Octans (§ V). Extensive evaluation results show that Octans can achieve reasonably prediction results for different NFs (2.3% prediction error on average) and different numbers of SFCs with varied lengths (2.6% prediction error on average). Moreover, Octans can improve the aggregate performance comparing to two alternative placement mechanisms by 26.7% to 51.8%. Finally, Octans has a high chance (58%$\sim$70%) to find an optimal solution in a short time (§ VI).

II. Related Work and Octans Novelty

This section summarizes state-of-the-art research on SFC placement in NFV, optimal thread scheduling in many-core systems, as well as works that touch upon the problem of optimal NF placement on many-core systems. However, to the best of our knowledge, Octans is the first to formally model the optimization problem, thoroughly study and model performance affection between NFs in many-core systems, and propose an efficient algorithm to solve the problem.

SFC Placement in NFV. Many efforts have been devoted to NF and SFC placement in NFV. Moens et al. [8] presented an Integer Linear Programming (ILP) model to minimize the number of used server in a service provider network. Cohen et al. [9] proposed a near-optimal algorithm to minimize the hop distance between a client and the NFs that process client's packets, as well as the setup costs of NFs. Kuo et al. [10] jointly considered NF placement and chaining across servers to better utilize network resource. Li et al. [16] focused on providing guaranteed performance for NFs by placing NFs in the right server. Sang et al. [17] aimed at minimizing the total number of NFs for packet processing. Ma et al. [18] focused on minimizing the maximum link load in the network with respect to traffic changes among NFs.

However, all existing works optimized SFC placement by mapping NFs to the right *servers*. Regarding SFC placement within a server, above works treated all CPU cores as equal, and did not consider the widely adopted NUMA architecture in modern many-core systems. In contrast, Octans studies the problem of placing one or multiple SFCs in a many-core system, in order to maximize aggregate throughput of all SFCs in a server. Octans deeply understands NF performance with respect to cross-node remote memory access and intra-node resource contention. Therefore, Octans could work with existing works to map right NFs to right *CPU cores*.

Optimal thread scheduling in many-core systems. Optimizing thread-to-core placement to maximize task execution performance in a many-core system has been extensively studied in the system and architecture community [11]–[14]. The key idea of above works is to measure the system performance metrics (e.g., LLC misses and memory load) at runtime, and *dynamically re-schedule* co-locating threads to different cores for optimal thread-to-core mapping. For example, Zhuravlev et al. [11] proposed dynamically scheduling co-locating threads according to the change of LLC misses.

Rao et al. [13] included more metrics such as data locality and sharing overhead , and converted them into a unified parameter for optimal VM scheduling in a shared server.

A natural question is *whether above solutions can be directly adopted to solve the SFC placement problem.* Our answer is *no*. Above solutions achieve optimal thread execution performance by frequently migrating threads across cores or nodes. While threads are light-weight and easy to migrate, most NFs in NFV are stateful and suffer from significant performance overhead during NF migration [15], [19]. Moreover, to achieve high performance, advanced NFV systems place NFs on dedicated CPU cores that cannot be scheduled by the operating system. This makes it difficult to migrate an NF among cores. Therefore, `Octans` maps NFs to cores by designing a *static placement* mechanism [20]. `Octans` thoroughly investigates and models NF performance in a many-core system and proposes an efficient placement algorithm to maximize aggregate SFC throughput.

SFC placement in many-core systems. Some recent researches [21]–[23] have revealed that placing NFs on different cores in a NUMA system could result in different performance. Sieber et al. [22] revealed the problem but did not present a solution for optimal placement. Wang et al. [21] presented a *locality-first-mapping* algorithm by placing an entire SFC in one node to avoid cross-node overhead, but the impact of intra-node contention was not considered. Hu et al. [23] investigated how the performance of *pipelined* software components varies when they are placed on different cores, and took NFV as an example. However, their solution is also based on dynamically scheduling, which could be impractical in real-world NFV systems as we discussed above.

III. Problem Formulation and Challenges

In this section, we first formulate the problem with a NLIP model. Then we identify the key factors for problem solving as evaluating NF performance drop index, and introduce the challenges in retrieving this parameter in many-core systems.

A. Formulation of the Optimal Placement Problem

Placement requirement of SFCs. Placement requirement of an SFC is usually described as an ordered sequence of NFs in the chain. Assume there is a set of SFCs S that require to be deployed, each of which is associated with an array of chained NFs $e(i)$, where $i \in S$.

A many-core system. Commonly in a many-core system, there are multiple nodes numbered incrementally and equipped with identical amounts of CPU cores. We generalize a many-core system with K nodes and each node has C cores.

Performance decomposition. We set a binary variable x_{ij}^k to indicate whether NF j of chain i is located on node k. When this NF runs without contention, the performance it can achieve is referred to as *ideal performance* and is defined as P_{ij}^k. As introduced in some work [5], [16], P_{ij}^k of each NF can be measured by placing the NF on different nodes since no contention exists. Furthermore, when an NF co-locates with other NFs in node k, its performance is referred to as *interfered performance* and is defined as ϕ_{ij}^k. It is obvious that the interfered performance of an NF is a reduced value of ideal performance. Thus, we define a performance drop index (denoted by λ_{ij}^k) to relate these two variables for the NFs that co-locate on node k (shown in Eqn. 2)

Similar to an end-to-end system, the processing capacity of an SFC is determined by the *bottleneck element* in the chain [24], i.e., the NF with the lowest performance. Hence, the objective is to maximize the aggregate performance across all required SFCs when deploying, which is formulated as,

$$max \sum_{i \in S} \min_{j \in e(i)} \{\phi_{ij}^k\} \tag{1}$$

where

$$\phi_{ij}^k = \sum_{k \in K} x_{ij}^k \cdot P_{ij}^k \cdot (1 - \lambda_{ij}^k), \quad \forall i \in S, j \in e(i) \tag{2}$$

s.t.

$$\sum_{k \in K} x_{ij}^k \leq 1, \quad \forall k \in K, i \in S, j \in e(i) \tag{3}$$

$$\sum_{i \in S, j \in e(i)} x_{ij}^k \leq C, \quad \forall k \in K \tag{4}$$

More specifically, Eqn. (2) describes the relationship between ideal performance and interfered performance. Constraint (3) prevents any NF from being repeatedly deployed. Constraint (4) specifies the capacity of a many-core system during deployment.

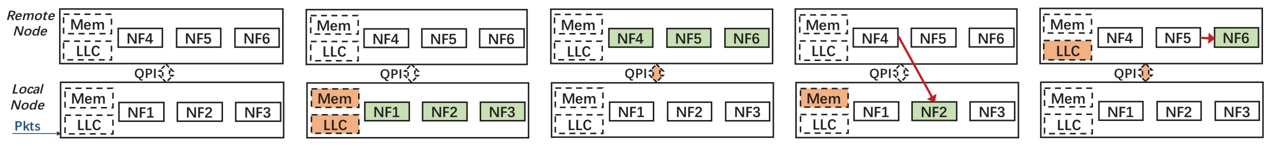
B. Key Factors for Problem Solving and Challenges

Problem formulation gives us a solid starting point for finding optimal placement for S. However, a critical parameter, i.e., λ_{ij}^k needs to be predicated for problem solving. However, predicting performance drop in the NFV context is not-trivial due to the following two challenges.

The heterogeneity of NFs: As introduced in § II, NFs are usually stateful and deployed on dedicated cores, which reveals that the placement should be static to avoid distractions from other complexities, such as state migration and careful packet buffering design. For the prediction of static placement, the inputs we have are only the types and counts of NFs (i.e., which NFs and how many of them) that might co-locate. However, it is tricky to adopt these inputs directly for the prediction model since they are not quantifiable. Some work [5] shows that we can use a manually analyzed and calculated system-level performance metric value (i.e., LLC references per packet) to quantify an NF. However, manual analysis and metric value calculation could be burdensome and platform dependent. Furthermore, this problem could be worse in the NFV context because NFs are usually heterogeneous and diverse.

SFC complicates the prediction: Performance drop of an application (NF) in an interference environment is usually related to the shared resources contention and its competitors [11], [13]. However, the chain in an SFC can change the shared resources of an NF, in contrast to which NFs are separated.

For example, consider six co-locating NFs on two nodes (the nodes where packets come in are usually called local nodes,



(a) An placement example on two nodes (b) Shared resources on local node (c) Shared resources on remote node (d) Shared resource changes due to cross-node chaining (e) Shared resource changes due to cache-prefetching chaining

Fig. 3: Different shared resources in different NF and SFC configurations, which are highlighted with orange-red.

Metric	Description
CPU_CLK_UNHALTED	Clock cycles when not halted
INST_RETIRED	Number of instructions retired
RESOURCE_STALLS	Cycles during which resource stalls occur
LLC_MISSES	Cache misses in LLC
L2_MISSES	Cache misses in Level 2 cache
IPC	Instructions per CPU cycle

TABLE I: A non-exhaustive list of commonly used metrics, which can be profiled by existing tools such as OProfile [25] and Intel PCM [26].

others are relatively remote nodes.) as shown in Fig. 3(a). For separated NFs, of which **(i)** in the local node can be considered as sharing LLC between NF1∼NF3 and main memory controller between NF1∼NF6 (Fig. 3(b)), and **(ii)** in the remote node can be considered as sharing QPI between NF4∼NF6 (Fig. 3(c)). We can see that all NFs in the same node have the same type of shared resources and competitors, thus one general model for these NFs might be accurate [5]. However, this model could be inaccurate when NF chaining is introduced. Fig. 3(d) shows that **(iii)** *cross-node chaining* between two NFs can change shared resources of the subsequent NFs (i.e., NF2) in a local node to be more stressed on main memory controller and almost no LLC benefit, which could incur performance drop. Moreover, Fig. 3(e) shows that **(iv)** *cache-prefetching chaining* can change shared resources of the subsequent NFs (i.e. NF6) to be less competing and benefit the performance improvement from packet memory cache hits in the remote LLC.

Therefore, we can see that more specific considerations are required in the prediction model due to the changes on shared resources and competitors introduced by SFC.

Octans. To address above problems, we propose Octans. For performance prediction, Octans takes the performance metric of each NF as the input for the prediction model as well as considering the specific features from SFC. Moreover, Octans adopts a heuristic-based placement algorithm to search optimal or near-optimal solutions to meet the requirement of online deployment. Next, we will introduce the design of Octans.

IV. Octans Design

In this section, we present the design of Octans. We first show how to automatically find performance metrics for NFs even they are heterogeneous, and what metrics is used in Octans (§ IV-A). Then, we introduce the performance prediction model (§ IV-B). Finally, we present the online placement algorithm in Octans (§ IV-C).

A. Relating NF with Performance Metrics

As mentioned in § III, the type of NFs cannot be used directly as the input of a prediction model. Instead, we should find some performance metrics to describe an NF. Moreover, as Table I shows, many potential system-level metrics can be adopted [13], [27]. Therefore, we should formally and automatically detect the appropriate metrics. To achieve this goal, we first identify the requirements these metrics should meet.

R1: The metric value should not vary even with contention. Since the metric we try to find should be able to describe an NF, its value should not vary when co-locating with other NFs. For example, if a metric value is measured as v_0 when an NF runs solely, all values ($v_1 \sim v_6$) measured when co-locating with other 1∼6 NFs should be near to v_0.

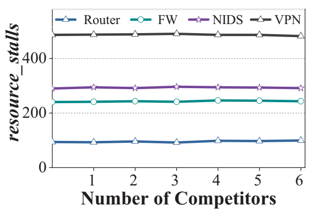
R2: The metric value should be sensitive to NF changes. This describes an intuition that different NFs should have different metric values. Here, we make a mild assumption that the NFs with similar intrinsic properties such as program complexity can achieve similar performance. Therefore, if the measured performance and metric values of a set of NFs when they run solely are $(p^1, p^2, p^3, ...)$ and $(v^1, v^2, v^3, ...)$, and $p^1 \neq p^2 \neq p^3$, the metric values should be approximately regarded as $v^1 \neq v^2 \neq v^3$.

With the help of above two requirements, we can use a general but formal way to find metrics. We re-interpret the two requirements as,

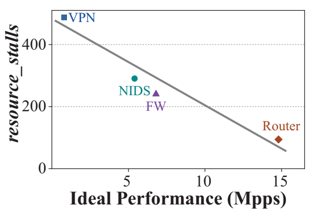
- *Near-zero variance of metric values when measured in competing environment:* Guided by R1, for each candidate metric, we calculate the variance (denoted by ρ_i^2) of its measured values when co-locating an NF with different types and different numbers of NFs, i.e.,

$$\rho_i^2 = \frac{\sum_{j=1}^{N}(m_{ij} - \mu_i)^2}{N} \tag{5}$$

  where, m_{ij} is the j-th sample of the value of metric i and μ_i is the mean value of all N samples. Moreover, all values used in this equation are normalized to [0,1].
- *Strong correlation between metric value and performance when measured in non-competing environment:* Guided by R2, there should exist strong correlation between metric value and performance when an NF runs solely (i.e., ideal performance). For concise expression, we abuse P_j to define the ideal performance of NF j. Some work [14], [27] has demonstrated that there is *linear correlation coefficient* between the value of some metrics and the performance. In our work, we follow their findings and assume this linear

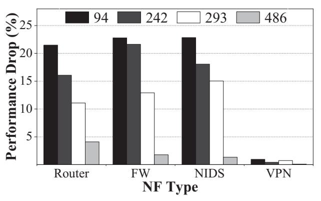


(a) Validation for R1

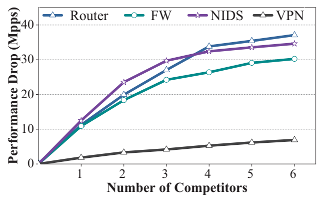


(b) Validation for R2

Fig. 4: Requirement validation for the metric *resource_stalls*.



(a) Performance drop with different NFs (represented by *resource_stalls*).



(b) Performance drop with different number of competitors.

Fig. 5: Performance drop with different NF type and number

correlation. We adopt Pearson Correlation Coefficient [28] to calculate the correlation (denoted as $r(P_j, m_i)$), i.e.,

$$r(P_j, m_i) = \frac{cov(P_j, m_i)}{\sigma_{P_j}\sigma_{m_i}} \tag{6}$$

where, $cov(P_j, m_i)$ is the cross-correlation of NF j between the ideal performance and metric i , and $\sigma^2_{P_j} = cov({P_j}^2)$, $\sigma^2_{m_i} = cov(m_i^2)$ are the variances of different measured values of performance and metric.

Metrics used in `Octans`. We measured a bunch of metrics and performance values according to Eqn. 5 and Eqn. 6. The candidate metrics we used are from two performance metric profiling tools, i.e., OProfile [25] and Intel Performance Counter Monitor (PCM) [26]. Table I shows a subset of these metrics. We empirically set $\rho_i^2 \leq 0.01$ as *near-zero variance* and $|r(P_j, m_i)| \geq 0.9$ as *strong correlation*. Metrics that meet both conditions are considered as appropriate ones.

Experiments based on sampled data reveal that the appropriate metric is *resource_stalls*. Note that we only choose the best metric for easier modeling. Nevertheless, we find that it is enough for our model to achieve accurate prediction (§ VI-B). Furthermore, we check whether this metric meets the two requirements defined before. Fig. 4(a) shows that for all four NFs, *resource_stalls* remains almost unchanged as the number of competing NFs increases, which meets the first requirement. Fig. 4(b) shows that *resource_stalls* and the ideal performance have near linear relationship, which can meet the second requirement. Therefore, we choose *resource_stalls* as the performance metric, and use it as the input of our prediction model.

B. Prediction Modeling

Preliminary analysis of the prediction model. Contention to competing resources is the root cause of performance degradation for co-locating NFs in a shared server [5], [13], [27]. According to observations from Fig. 3, we classify competing resources into two types of factors that could cause performance drop including LLC misses and memory contention (including memory controller and QPI). Moreover, we utilize the system-level metric of NFs (i.e., *resource_stalls*) to predict performance drop index λ (we reuse λ here for brevity). Next, we first show how to quantify the competing level of an NF under one type of competing resource. Based on that, we provide the model for λ. Finally, we refine the model to improve prediction accuracy according to SFC characteristics.

Calculating the competing level of an NF under one type of shared resource. We define a competing level function f to map the metric value of competing NFs to performance drop. To explore this function, we first check the relationship between the competing level and metric value changes, i.e., how the competing level varies when the metric value changes. It is hard to directly capture the change of competing level. Instead, we use the performance drop to imply it. Fig 5(a) shows that for different NFs, a smaller *resource_stalls* incurs higher performance drop, i.e., higher competing level. From this observation, we define a property for this function,

- *Property 1:* The function shows *negative correlation* to *resource_stalls*.

Furthermore, we check that how the competing level varies with the number of NFs changes, i.e., aggregate *resource_stalls*. Fig. 5(b) shows that with the number of competing NFs increases, the performance drops more, i.e., higher competing level. Moreover, we can observe that the increasing gradient of performance drop becomes small. Therefore, we define another property for this function,

- *Property 2:* The function shows the *increment property and decreasing gradient* with aggregate *resource_stalls* increases.

With the help of above two properties, we can approximately describe the competing level function as,

$$f(\{x_1, x_2, ..., x_n\}) \approx \sum_{1 \leq k \leq n} \frac{1}{x_k} \tag{7}$$

It can be easily proved that the function in Eqn. 7 meets the two defined properties. Although this function seems simple and intuitive, our evaluation (§ VI) shows it is accurate enough to describe the competing level in the prediction model.

Modeling performance drop index under multiple competing resources. According to [27], in an interference environment, we could model the performance drop as a linear function of different competing resources . Therefore, we define λ as,

$$\lambda = \alpha \cdot f(\{m^i | i \in NF_{LLC}\}) + \beta \cdot f(\{m^i | i \in NF_{mem}\}) + C \tag{8}$$

where, NF_{LLC} and NF_{mem} indicate the competitors on LLC and memory. α, β and C are parameters to aggregate the effect of competing level of LLC and memory. Since every NF could be placed all nodes, there should be distrinct parameter groups for the local and remote nodes due to the asymmetric cost of LLC misses and memory access [12]. Thus, we define (α_l^i,

β_l^i, C_l^i) for the case that NF i is deployed in the local node, and (α_r^i, β_r^i, C_r^i) for the case of remote nodes. Here, we adopt the same parameters for all remote nodes. It is because that all remote nodes use the same interconnection (i.e., QPI) to access the memory on a local node.

Fine-tuning prediction model under SFC characteristics. As mentioned in § III, SFC will complicate the prediction model since different chaining methods (e.g., cross-node) could change the competitors of an NF. To address this problem, we first introduce a general model for the NF in a local or remote node, then tune this model to address the challenge imposed by SFC for accurate prediction.

(1): A general prediction model. Let $\{m^l\}$ and $\{m^r\}$ denote the measured *resource_stalls* of those NFs locate in a local node (NF_l) and a remote node (NF_r). Consider an NF in a local node shown in Fig. 3(b), it will compete for LLC with the NFs in the local node, and compete for memory controller with all NFs inside the same server. For an NF in a remote node, we set its LLC misses to "1", i.e., no cache hit to packet memory. This is because it cannot access the cache on the local node. Moreover, since the bandwidth of QPI is lower than the main memory controller, we consider QPI as the competing resource and the NFs in the remote node as the competitors. Therefore, we construct the model as,

$$\lambda^i = \begin{cases} \alpha_l^i \cdot f(\{m^l\}) + \beta_l^i \cdot f(\{m^l\} \cup \{m^r\}) + C_l^i, \ \forall i \in NF_l \\ \alpha_r^i \cdot 1 + \beta_r^i \cdot f(\{m^r\}) + C_r^i, \ \forall i \in NF_r \end{cases} \tag{9}$$

(2): Tuned model for SFC. As introduced in § III, two chaining methods should be considered. The first one is *cross-node chaining*. As shown in Fig. 3(d), for an NF (i.e., NF2) in the local node, its upstream NF is in the remote node. This makes the NF has no chance to read packets from its local LLC. Thus, we set its cache misses to "1", i.e.,

$$\lambda^i = \alpha_l^i \cdot 1 + \beta_l^i \cdot f(\{m^l\} \cup \{m^r\}) + C_l^i \tag{10}$$

The second method is *cache-prefetching chaining*. As shown in Fig. 3(e), for an NF (i.e., NF6) in the remote node, we do not regard it as other NFs (e.g., NF4) that have no chance to read packets from LLC. This is because that its upstream NF (i.e., NF5) could load packets into remote LLC as mimicking a cache prefetcher [29]. As a result, NF4 has the chance to access packet in remote LLC and no longer to be "1" for LLC misses. Moreover, We consider the NFs in the remote node as its LLC competitor. Thus, we tune the model for *cache-prefetching chaining* as,

$$\lambda^i = \alpha_r^i \cdot f(\{m^r\}) + \beta_r^i \cdot f(\{m^r\}) + C_r^i \tag{11}$$

Moreover, due to NFs in a cache-prefetching chain share the same piece of LLC, we regard these NFs as only one competitor to other NFs for LLC sharing. Currently, we use the first NF in this chain to calculate competing level.

One can see that if measuring samples are given, including the placement of SFCs and performance drop index of each NF, it is easy to approximate the parameters (α, β, C) for each NF. In `Octans`, we use a fast learning algorithm, Least Mean Squares (LMS), to approximate them.

Algorithm 1: Online Placement Algorithm

```
input  : $(S)$ - SFC sets
output : $X$ - Placement of each NF on the SFCs
1  // 1: Initial placement
2  $X_{init} \leftarrow$ Binary-search all possible placement for $s \in S$ with
     Constraint 3 and 4.
3  // 2: Move some subchains in the remote nodes to local node
4  $Perf_{opt} \leftarrow 0$ // The optimal performance
5  foreach $x \in X_{init}$ do
6      $(S_{local}, S_{remote}) \leftarrow$ The SFCs placed in the local and remote
         node from $x$
7      Update_Placement $(S_{local}, S_{remote})$
8      DFS$(S_{local}, S_{remote})$

9  function DFS $(S_{local}, S_{remote})$
10     foreach $s \in S_{remote}$ do
11         $NF_{bottleneck} \leftarrow \min\{NF \in s\}$ // The NF with lowest
             performance
12         $s_{subchain} \leftarrow s[0 : NF_{bottleneck}]$ // The subchain
13         if Enough cores for $s_{subchain}$ in the local node then
14             $(tmp\_S_{local}, tmp\_S_{remote}) =$
                 $(S_{local} + S_{subchain}, S_{remote} - S_{subchain})$
15             Update_Placement$(tmp\_S_{local}, tmp\_S_{remote})$
16             DFS$(tmp\_S_{local}, tmp\_S_{remote})$

17 function Update_Placement $(S_{local}, S_{remote})$
18     $Perf_{prediction} \leftarrow$ Performance prediction from Eqn. 9, 10 and
         11
19     if $Perf_{opt} < Perf_{prediction}$ then
20         $Perf_{opt} = Perf_{prediction}$
21         $X = \{S_{local}, S_{remote}\}$ // updating placment
```

C. Online Placement Algorithm

So far, we have constructed the prediction model for the performance drop index in Eqn. 2, which makes our formulation solvable. A naive approach to finding optimal solution is Brute-force search, but it can be expensive. Since one can see that its time complexity is $O(K^N)$ (K is the total number of available nodes and N is the total number of NFs in all SFCs). However, the network operator usually needs fast response for placement requests [6], which implies a short time for solution searching. Furthermore, our problem is a Multidimensional Assignment Problem (MAP) (an NF can be assigned any one of multiple nodes), which is a known NP-complete problem [20]. Therefore, there is no polynomial-time algorithm to find an optimal solution. Instead, we propose a heuristic-based online placement algorithm to find the optimal or near-optimal solution.

Our heuristic algorithm follows two major steps: (1) *initial placement: chain-based search.* Instead of searching placement for all NFs, we first try to place an entire chain on the same node. It is based on an important observation that a cache-prefetching chain always has lower performance drop against a cross-node chain from Eqn. 10 and Eqn. 11. This is because that the output of $f(\{m^l\})$ and $f(\{m^r\})$ is always smaller than 1; and (2) *moving subchains in a remote node to the local node.* A common sense is that an NF in the local node usually has higher performance than in a remote node due to benefiting from lower memory access latency (i.e., iMC is faster than QPI). Hence, to improve the chance to find an optimal solution based on the initial placement, we try to move some subchains of entire chains in the remote node to the local

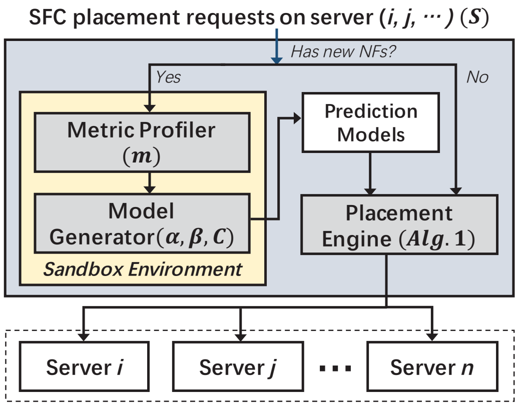

SFC placement requests on server $(i, j, \cdots)$ (S)
Has new NFs?
Yes
No
Metric Profiler (m)
Prediction Models
Model Generator(α, β, C)
Placement Engine $(Alg. 1)$
Sandbox Environment
Server i
Server j
Server n

Fig. 6: Architecture and workflow of Octans.

node, and the split point in an entire chain to form a subchain is the bottleneck-NF (i.e., with the lowest performance).

We show the online placement algorithm in Algorithm 1. It first produces the initial placement via binary-search for all possible placements with entire SFCs across all nodes (line 2). Then, iterate each initial placement to find potential placement with higher aggregate performance by moving some subchains from a remote node to the local node (lines 5-8). We adopt deep-first search (DFS) to find the bottleneck-NF for each SFC (lines 9-16), meanwhile update the potential optimal solution if the moving operation produces higher performance (lines 17-21).

V. Octans Architecture and Workflow

According to the design in § IV, we present the architecture and workflow of Octans as shown in Fig. 6. Octans takes SFC placement requests on a specific server (i.e., S) as the input, and checks whether or not existing new NFs in inputing SFCs. If existed, they are sent to a *sandbox environment* to calculate the metric value (i.e., *resource_stalls*) and generate performance model, which is supported by two major modules, *Metric Profiler* and *Model Generator*; If not, requests are immediately sent to *Placement Engine*, to calculate placement solution. Next, we show the detail of these three modules.

Metric Profiler: Metric Profiler automatically profiles *resource_stalls* for every NF. In current implementation, it uses OProfile [25] to measure the count of *resource_stalls*. Since Octans focuses on the performance in throughput, we generate 10 Gbps traffic (this rate can be set) with minimum size packets (i.e., 64B). Metric Profiler records the total number of packets it processed (excluding dropped packets) and the total count of *resource_stalls* after a period of running, then calculates *resource_stalls* per packet as the metric value.

Model Generator. Model Generator generates the three parameters (α, β, C) for every NF. First, it runs each NF solely to record the performance as ideal performance (i.e., P_{ij}^k in Eqn. 2). Second, it co-locates this NF with different types and numbers of NFs (including itself) to measure the interfered performance ϕ_{ij}^k, meanwhile records the performance drop index with calculation $\frac{P_{ij}^k - \phi_{ij}^k}{P_{ij}^k}$ as sampled data. After sampling enough data (this might take a long time, but it can be completed offline), this generator adopts LMS algorithm to approximate the parameters. Finally, it updates the record of *Prediction Models*, where stores the prediction model of every NF.

Placement Engine. Placement Engine runs the online placement algorithm (§ IV-C). It takes the placement requests (i.e., S) as input, and retrieves the model of each NF from *Prediction Models*. After an optimal or near-optimal solution is found, it launches the required NFs in SFC requests on assigned CPU cores of target servers.

VI. Implementation and Evaluation

A. Implementation and Experiment Setup

Implementation. We have implemented a prototype of Octans. The infrastructure we used to run NFs and SFCs is built on a high-performance NFV platform, OpenNetVM [2]. The NFs we used include *IPv4Router* (Router), *Firewall* (FW), *NIDS*, and *VPN*. The three modules in Octans are written in the Python language.

Experiment setup. We run Octans and OpenNetVM on the same server, which is a two-node server that is equipped with two Intel Xeon E5-2650 v4 CPUs (2.20 GHz, 12 physical cores), 128GB total memory, and two dual-port 10G NICs (Intel X520-DA2, 40 Gbps in total). We use DPDK Pktgen [30] to generate traffic, which runs on another server that has the same configuration as the previous one. Both servers run Ubuntu 14.04 (with kernel 3.16.0-30), and DPDK version 18.02. The two NICs are located on Node-0, hence we treat Node-0 as the local node and Node-1 as the remote node. We allocate 1 CPU core to Octans, 1 core to the openNetVM manager, and 4 cores for networking I/O and packet forwarding. Herefore, we have 6 available cores in the local node and 12 available cores in the remote node.

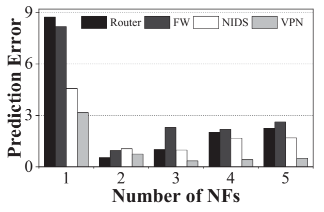
We evaluate Octans with the following goals.

- Demonstrate that Octans can achieve accurate performance drop prediction for a wide range of NFs and SFCs.
- Demonstrate that Octans can improve aggregate performance by searching optimal or near-optimal placement solution, compared with two alternative placement solutions.
- Demonstrate that Octans can search the solution in a short time, and has reasonable chance to find the optimal solution.

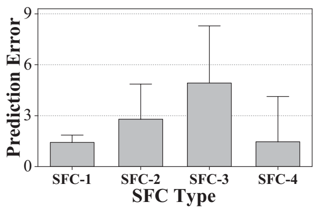
B. Prediction Accuracy.

To demonstrate the prediction accuracy of performance drop for NFs and SFCs, we compare the performance drop between our prediction value ($\lambda^{predicted}$) and the measured value ($\lambda^{measured}$) by deploying them into our testbed. We calculate the prediction error as $\frac{|\lambda^{measured} - \lambda^{predicted}|}{\lambda^{measured}}$.

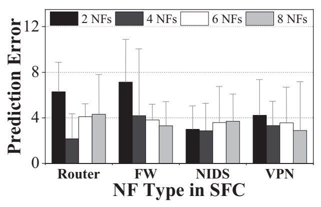
Prediction accuracy for different NFs. We first evaluate the prediction accuracy for separated NFs when co-locating with the competitors. We use a synthetic NF (ideal performance between NIDS and Firewall) as the co-locating NF (competitor) for our evaluated NFs. Fig. 7(a) shows the prediction error of four NFs when they co-locate with different number of competitors. From this figure, we can see reasonablely accurate results, an average of 2.9%, 3.3%, 2.0% and 1.0%



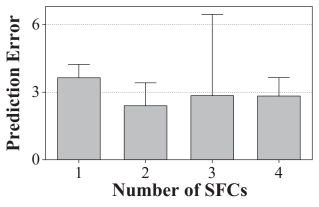
(a) Prediction error on different NFs.



(b) Prediction error on different SFCs.



(c) Prediction error on different length of SFCs.

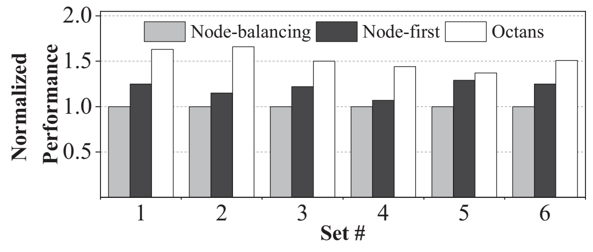


(d) Prediction error on different number of SFCs.

Fig. 7: Prediction accuracy.

Set #	SFCs
1	(Router, Firewall, NIDS); (Router, NIDS); (VPN, Firewall)
2	(Router, Firewall, VPN); (Router, Firewall); (Firewall, NIDS)
3	(Router, Firewall, NIDS); (Router, Firewall, VPN); (NIDS, Firewall)
4	(Router, Firewall, NIDS); (Firewall, NIDS, VPN); (Firewall, Router)
5	(Router, Firewall, NIDS); (Router, Firewall, VPN); (Firewall, NIDS, VPN)
6	(Router, Firewall, NIDS, VPN); (Router, Firewall, NIDS); (NIDS, Router)

(a) The SFC sets used in this evaluation.



(b) Aggregate performance improvement of different SFC sets.

Fig. 8: Improvement of aggregate performance.

deviation from the measured performance drop for Router, Firewall, NIDS and VPN, respectively. Also, we can observe that with the number of competitors increases, the prediction results become more accurate. For example, the prediction error reduces from 8.7% to 2.3% for the Router. This is because the gradient of performance drop becomes smaller with the competing level increases, and our prediction model can capture this feature.

Prediction accuracy for different SFCs. We then evaluate the prediction accuracy for four customised SFCs and mark them: SFC-1 (Router, Firewall, NIDS, VPN), SFC-2 (Firewall, Router, NIDS, Firewall), SFC-3 (NIDS, Router, Firewall, NIDS), and SFC-4 (Router, NIDS, Router, Firewall). We randomly generate 20 types of placement for every SFC. Fig. 7(b) shows the average prediction error of them. We can see that all of them have reasonably accurate results, and the worst-case of average error (SFC-3) is less than 5.1%.

Effect of the varied length of SFC. To show more reliable prediction results, we evaluate whether the length of SFC has the potential to incur higher prediction error. We use four SFCs, and NFs in each SFC are the same. For different lengths of SFC, we randomly generate 20 types of placement. Fig. 7(c) shows the prediction error for the four SFCs with different lengths. We can still see reasonably accurate results, which range from 2.2% to 6.3%, 3.3% to 7.1%, 2.9% to 3.7%, and 2.9% to 4.2% for these SFCs, respectively.

Effect of the varied number of co-locating SFCs. Finally, we evaluate the effect of the varied numbers of co-locating SFCs on prediction results. We use an SFC with three NFs (Router, Firewall, NIDS), and increase the number of co-locating SFCs from 1 to 4. We still generate 20 types of placement for each testing scenario. Fig. 7(d) shows that the prediction error varies from 2.4% to 3.6%, which is also reasonably accurate result.

C. Improvement of aggregate performance

To demonstrate that `Octans` can improve the aggregate performance by finding optimal or near-optimal SFC placement, we compare it with two alternative placement mechanisms that can be used in current systems: (1) *node-balancing placement:* it places NFs by dividing them to all nodes for balancing the core utilization on each node; and (2) *node-first placement:* it tries to place NFs on the local node first until no core is available, then places NFs on other nodes. We use 6 sets of SFCs, each with different SFC requests (Fig. 8(a)).

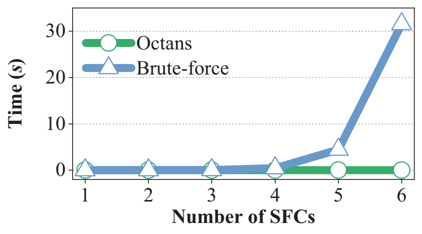
Fig. 8(b) shows the normalized performance of `Octans` and the two alternative mechanisms. We can see that in SFC sets, `Octans` can achieve more aggregate performance than Node-balancing placement by an average of 51.8% and ranging from 37.4% to 66.3%. Also, comparing with *node-first placement*, `Octans` can achieve more aggregate performance by an average of 26.7% and ranging from 6.2% to 45.1%.

D. Efficiency of Online Placement Algorithm

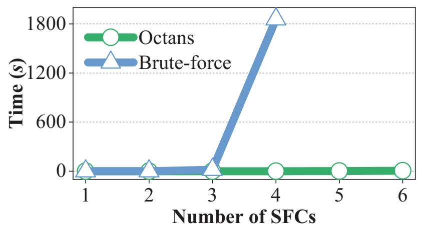
Time cost of solution search. We compare the placement algorithm in `Octans` to the naive *brute-force search* algorithm. We use an SFC with 3 NFs and the varied number (1$\sim$6) of SFCs as the input for each algorithm. In each set of SFCs, we randomly replace NFs and repeat 100 times. A many-core system usually has 2 or 4 nodes [31]. Therefore, we also evaluate the effect of the number of nodes (setting 10 cores in each node) on calculation time. We only allocate one CPU core (2.2 GHz) to run the algorithm.

Fig. 9(a) shows that in a 2-node server, even in the worst-case (18 NFs in 6 SFCs), our algorithm can find a solution with an average time of 0.017s, which takes 1853x less time than the *brute-force search algorithm* (31.5s). Also, in a 4-node server as shown in Fig. 9(b), `Octans` can find a solution with an average time of 5.3s when in the worst-case, but the *brute-force search* needs to spend more than 1858s (we use the case with 4 SFCs as this value due to the long calculation time of 5 and 6 SFCs).

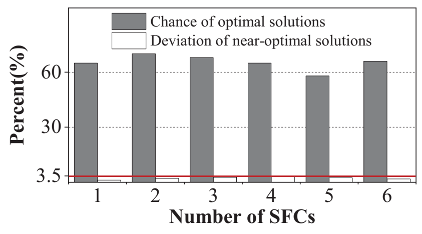
The chance to find optimal solutions. Our algorithm is heuristic based, hence it cannot guarantee an optimal solution. We evaluate the chance that our algorithm can find optimal solutions, and if a near-optimal solution can be found, and what the deviation it is. The optimal solution is found by



(a) Time cost with different number of SFCs in a 2-nodes server.



(b) Time cost with different number of SFCs in a 4-nodes server.



(c) Optimality gap of our heuristic algorithm.

Fig. 9: The time cost and optimality gap of our algorithm.

brute-force search algorithm (but taking long time), and we run the two algorithms with a simulated 2-nodes server. The SFCs and NF generation is similar to the previous evaluation, and we also repeat each set 100 times. Fig. 9(c) shows `Octans` has a (58%∼70%) chance to find an optimal solution across different numbers of SFCs. Moreover, we observe that when even the solutions are near-optimal, they only have an average of 1.2%∼3.5% deviation from the optimal solutions.

VII. Conclusion

We have presented `Octans`, an NFV orchestrator that searches optimal or near-optimal placement for SFCs in a many-core NFV system. Starting with an NLIP model, `Octans` first constructs an accurate model to predict an unknown parameter in this model with automatically identified metric of NFs as the input. Then, it adopts an online placement algorithm to quickly find a solution for the placement requests. Our evaluation built upon openNetVM shows that `Octans` provides accurate prediction results, significantly improving the aggregate performance in the system, and has a high chance to find optimal solutions in a short time.

VIII. Acknoledgement

We would like to thank the anonymous reviewers for their thoughtful and insightful comments. This work is supported by the National Key R&D Program of China (2017YFB0801701), and the National Science Foundation of China (No.61872426). Jun Bi (junbi@tsinghua.edu.cn) is the corresponding author.

References

[1] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: a framework for nfv applications," in *SOSP*, 2015.
[2] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood, "Opennetvm: A platform for high performance network service chains," in *HotMiddlebox*. ACM, 2016.
[3] OpenStack, "Openstack," 2018. [Online]. Available: https://www.openstack.org/
[4] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A case for numa-aware contention management on multicore systems," in *ATC*. USENIX, 2011.
[5] M. Dobrescu, K. Argyraki, and S. Ratnasamy, "Toward predictable performance in software packet-processing platforms," Tech. Rep., 2012.
[6] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker, "Resq: Enabling slos in network function virtualization," in *NSDI*, 2018.
[7] C. Zeng, F. Liu, S. Chen, W. Jiang, and M. Li, "Demystifying the performance interference of co-located virtual network functions." INFOCOM, 2018.
[8] H. Moens and F. De Turck, "Vnf-p: A model for efficient placement of virtualized network functions," in *CNSM*, 2014, pp. 418–423.
[9] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, "Near optimal placement of virtual network functions," in *INFOCOM*. IEEE, 2015.
[10] T.-W. Kuo, B.-H. Liou, K. C.-J. Lin, and M.-J. Tsai, "Deploying chains of virtual network functions: On the relation between link and server usage," *IEEE/ACM Transactions on Networking*, no. 99, pp. 1–15, 2018.
[11] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *ASPLOS*, vol. 45, no. 3. ACM, 2010, pp. 129–142.
[12] B. Lepers, V. Quéma, and A. Fedorova, "Thread and memory placement on numa systems: Asymmetry matters." in *USENIX ATC*, 2015.
[13] J. Rao, K. Wang, X. Zhou, and C.-Z. Xu, "Optimizing virtual machine scheduling in numa multicore systems," in *HPCA*. IEEE, 2013.
[14] M. Liu and T. Li, "Optimizing virtual machine consolidation performance on numa server architecture for cloud workloads," in *ISCA*. IEEE, 2014, pp. 325–336.
[15] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," in *SIGCOMM*, vol. 44, no. 4. ACM, 2014, pp. 163–174.
[16] Y. Li, L. T. X. Phan, and B. T. Loo, "Network functions virtualization with soft real-time guarantees," in *INFOCOM*. IEEE, 2016, pp. 1–9.
[17] Y. Sang, B. Ji, G. R. Gupta, X. Du, and L. Ye, "Provably efficient algorithms for joint placement and allocation of virtual network functions," in *INFOCOM*, 2017, pp. 1–9.
[18] W. Ma, O. Sandoval, J. Beltran, D. Pan, and N. Pissinou, "Traffic aware placement of interdependent nfv middleboxes," in *INFOCOM*, 2017.
[19] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes." in *NSDI*, vol. 13, 2013, pp. 227–240.
[20] Y. Jiang, X. Shen, C. Jie, and R. Tripathi, "Analysis and approximation of optimal co-scheduling on chip multiprocessors," in *PACT*, 2008.
[21] Y. Wang, "Numa-aware design and mapping for pipeline network functions," in *ICSAI)*. IEEE, 2017, pp. 1049–1054.
[22] C. Sieber, R. Durner, M. Ehm, W. Kellerer, and P. Sharma, "Towards optimal adaptation of nfv packet processing to modern cpu memory architectures," in *Proceedings of the 2nd Workshop on Cloud-Assisted Networking*. ACM, 2017.
[23] Y. Hu and T. Li, "Towards efficient server architecture for virtualized network function deployment: Implications and implementations," in *Micro*. IEEE Press, 2016, p. 8.
[24] Y. Zhang, W. Wu, S. Banerjee, J.-M. Kang, and M. A. Sanchez, "Sla-verifier: Stateful and quantitative verification for service chaining," in *INFOCOM*, 2017.
[25] OProfile, "Oprofile," 2018. [Online]. Available: http://oprofile.sourceforge.net/news/
[26] Intel, "Intel performance counter monitor," 2017. [Online]. Available: https://software.intel.com/en-us/articles/intel-performance-counter-monitor/
[27] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: managing performance interference effects for qos-aware clouds," in *EuroSys*. ACM, 2010, pp. 237–250.
[28] J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," in *Noise reduction in speech processing*. Springer, 2009.
[29] J. Collins, S. Sair, B. Calder, and D. M. Tullsen, "Pointer cache assisted prefetching," in *Micro*. IEEE Computer Society Press, 2002, pp. 62–73.
[30] I. D. Community, "Dpdk pktgen," 2018. [Online]. Available: http://pktgen-dpdk.readthedocs.io/en/latest/
[31] Dell, "Dell poweredge servers portfolio guide," 2018. [Online]. Available: http://www.dell.com/downloads/global/products/pedge/en/pedge-portfolio-brochure.pdf